Speeding Up Model Building for ECGA on CUDA Platform

Chung-Yu Shao
Department of Electrical Engineering
National Taiwan University
r00921046@ntu.edu.tw

Tian-Li Yu
Department of Electrical Engineering
National Taiwan University
tianliyu@cc.ee.ntu.edu.tw

ABSTRACT

Parallelization is a straightforward approach to enhance the efficiency for evolutionary computation due to its inherently parallel nature. Since NVIDIA released the compute unified device architecture (CUDA), graphic processing units have enabled lots of scalable parallel programs in a wide range of fields. However, parallelization of model building for EDAs is rarely studied. In this paper, we propose two implementations on CUDA to speed up the model building in the extended compact genetic algorithm (ECGA). The first implementation is algorithmically identical to original ECGA. Aiming at a greater speed boost, the second implementation modifies the model building. It slightly decreases the accuracy of models in exchange for more speedup. Empirically, the first implementation achieves a speedup of roughly 359 to the baseline on 500-bit trap problem with order 5, and the second implementation achieves a speedup of roughly 506 to the baseline on the same problem. Finally, both of our implementations scale up to 9,800-bit trap problem with order 5 on one single Tesla C2050 GPU card.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming - *Parallel programming*

General Terms

Algorithms, Performance, Experimentation

Keywords

CUDA, GPU, Estimation of Distribution Algorithms, ECGA, Model Building, Efficiency Enhancement

1. INTRODUCTION

The extended compact genetic algorithm (ECGA) [5] belongs to one of the evolutionary computations (ECs) called the estimation of distribution algorithms (EDAs) [1, 10, 8]. EDAs are stochastic optimization techniques that utilize the relations among genes to guide the search process via probabilistic models. Probabilistic models summarize the information of related genes from the promising candidate solutions. EDAs then generate offspring via sampling these models.

EDAs have been regarded as *competent* [4] that solve boundedly difficult problem in subquadratic number of function evaluations. However, for large-scale problems, the subquadratic number of function evaluations can still be time consuming. Therefore, efficiency enhancement techniques [18] are developed for large-scale, complex problems. Since ECs are inherently parallel, parallelization is a straightforward approach to enhance the efficiency [2].

Nowadays, the graphic processing unit (GPU) is a widely affordable parallel processor to solve massively parallel computation tasks. Since NVIDIA released the compute unified device architecture (CUDA) in 2007 [16], lots of scalable parallel programs have been developed in a wide range of fields. CUDA is not only a minimal extension of the C/C++ programming, but also a parallel computing platform that unifies graphics and general-purpose parallel computing applications. Developers write serial program that calls parallel *kernel* functions. When called, the kernel function is executed T times in parallel, where T is the number of CUDA threads defined by the programmer. The remaining portion of program is then executed sequentially on the central processing unit (CPU). Therefore, CUDA facilitates a heterogeneous computing between CPUs and GPUs. Although CUDA is friendly for a programmer familiar with C to start with, the performance of the program relies on the comprehension of several software and hardware properties, which are defined by CUDA's *compute capability*.

Researchers have implemented many ECs on CUDA, such as differential evolution [7], genetic algorithm (GA) [21, 11] and particle swarm optimization [13]. However, to the best of our knowledge, there are comparatively fewer researches related to EDAs with CUDA. Munawar *et al.* [12] proposed CUDA-based parallel Bayesian optimization algorithm (BOA). In their implementation, the Bayesian network construction in BOA [17] is implemented on CUDA, and the implementation speeds up a 192-bit test problem 13 times.

In this paper, we propose two CUDA-based implementations aiming at speeding up model building for ECGA. We first introduce a table-lookup method to speed up the process of counting the occurrences of schemata. This technique is then implemented on GPU with maximum utilization of

GECCO'13, July 6–10, 2013, Amsterdam, The Netherlands.

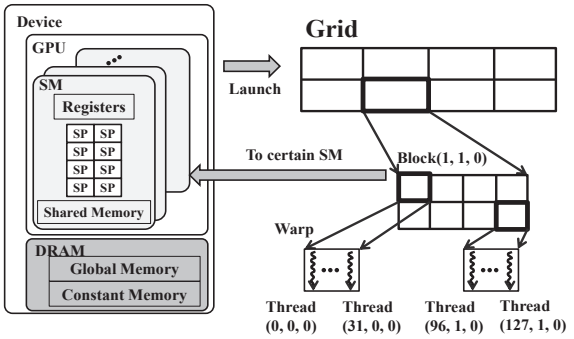


Figure 1: The GPU device lunches a grid with dimension (2, 4, 1) of thread blocks. In each thread block, there are 256 threads, which are divided into 8 warps.

available shared memory. Based on the first implementation, the second implementation modifies the greedy model-searching algorithm to obtain more speedup. The remainder of the paper is divided into five sections. In Sections 2 and 3, we describe the background of CUDA and ECGA in greater detail and review previous work on parallelism on ECGA. The proposed implementations are described in Section 4, and the empirical results are discussed in Section 5. Finally, we conclude this paper in Section 6.

2. GPU AND CUDA

In this section, the background of CUDA is discussed, followed by the programming model of CUDA. The CUDA design constraints are emphasized because these constraints affect how we design our algorithms. The section is mainly reorganized and summarized from [14, 15, 16].

2.1 Background

GPU was originally designed to support graphics rendering tasks. Specifically, a program for one thread drew one vertex or shaded one pixel fragment. Thousands of independent threads executed concurrently in a fine-grained, data-parallel sense on GPU. Because GPU was specialized for such highly parallel tasks with intensive computation, more transistors were devoted to data processing rather than caching and flow control.

The design made GPU well-suited to problems that could be expressed as data-parallel computations. The memory access latency could be hidden by many data elements that were executed at the same time. Traditionally, researchers had to explore a mapping between the non-graphics computations and the graphics rendering in order to utilize the resource of GPU. The need for general purpose computation on GPU hence motivated NVIDIA to develop CUDA.

2.2 CUDA Programming Model

Three key abstractions form the core of the CUDA programming model: (1) a hierarchy of thread groups, (2) shared memories, and (3) barrier synchronization. The hierarchy of thread groups enables programmers to control all the threads launched by kernel function, which is illustrated in Figure 1. A kernel executes across a set of threads parallely, and the set of threads is organized as a hierarchy of *grid* of *thread blocks*. Specifically, a grid is formed by several three-dimensional thread blocks. Each thread block contains a three-dimension of threads. Each thread in a given thread block has its unique thread index number. Threads in same thread block cooperate through the per-block shared memory and synchronize with __SYNCTHREADS(). Threads in different blocks cooperate through the global memory and synchronize through terminating the current kernel.

When a kernel launches, GPU automatically distributes the thread blocks to the streaming multiprocessors (SMs). SM creates, manages, schedules, and executes threads in *warp*, which is a group of 32 parallel threads. When given one or more thread blocks to execute, SM partitions each block to warps. Each warp is then scheduled by a warp scheduler for execution. A warp executes one common instruction at a time, which is called the single instruction multiple threads parallel programming model. The compute capability of the device defines the maximal number of resident warps on a SM. The resident blocks on a SM are divided into resident warps, and the resident warps are executed concurrently. When blocks terminate, GPU launches new blocks on the vacated SMs.

2.3 Design Constraints

There are many constraints for CUDA programming due to the GPU architecture. Below we describe three constraints in greater detail. The first constraint is the memory latency issue. As previously mentioned, threads in same block communicate through the on-chip shared memory. Due to the GPU architecture, the shared memory is expected to be a low-latency (few cycles) memory near each SM which provides high performance communication. In contrast, threads in different blocks or sequentially dependent grids communicate via the high-latency (hundreds of cycles) global memory. Nevertheless, if consecutive threads access consecutive memory addresses, all of the threads in a half-warp access the global memory at the same time. This process is called *coalesced memory* transaction.

The second constraint is the memory bandwidth. GPU has hundreds of cores that provide high arithmetic throughput. However, memory traffic in the global memory happens because we can't keep the input coming to GPU fast enough to sustain such high rates of computation. Specifically, when too many threads are acquiring data from the global memory, GPU might prevent all but few of threads from accessing. Many SMs are therefore idle. To evaluate the efficiency of how we utilize the limited bandwidth, the *compute to a global memory access* (CGMA) ratio is defined as the number of floating-point calculations performed for each access to the global memory within a region of a CUDA program. For example, the peak computation of NVIDIA GeForce 8800 GTX is 367 GFLOPS, and the memory bandwidth is 86.4GB/s. A program achieves the peak computation of this GPU until CGMA is above $\frac{367\times4}{86.4} \simeq 16.99$.

The last constraint with lower optimization priority is to maintain sufficient number of active threads per SM so that GPU can hide the memory latency. The ratio of the number of active warps per SM to the maximum number of possible active warps per SM is defined as the *occupancy* metric. This occupancy metric shows the degree to which we keep GPU busy. Unfortunately, a trade-off exists between occupancy and the resource of the shared memory per block. Table 2.3 indicates that to achieve full occupancy on a Tesla C2050

	GT8000	Tesla C2050
Compute capability	1.1	2.0
Shared memory size per MP (byte)	16K	48K
Maximum # resident blocks per SM	8	8
Shared memory when full occupancy	2K	6K

Table 1: The shared memory for 2 architecture of GPUs with 100% occupancy.

GPU card, one block only has 6,000 bytes of the shared memory. For a GPU card with lower compute capability, the amount of the shared memory per block is less.

To conclude, five policies are keys for high-performance CUDA program: (1) using the shared memory instead of the global memory, (2) using the global memory as coalesced as possible, (3) minimizing data transfer through the global memory between CPU and GPU, (4) keeping CGMA high, and (5) trying to keep GPU busy so that the latency can be hidden. For further information, please refer to [14, 15, 16].

3. ECGA

The section first describes EDAs and ECGA. ECGA, one of EDAs, models the population with the marginal product model. The model is decided by the combined complexity criteria that represents the description length of the model. The overall model building in ECGA and a previous work on ECGA with parallelism are described in greater detail.

3.1 Introduction to EDAs and ECGA

In the GA field, the term *linkage* refers to the relations among variables. A group of related variables can be seen as a *building block* (BB). Since Holland [6] addressed the importance of BBs, many different linkage-learning techniques have been developed for GAs. Studies have shown that losing linkage results in disruptive mixing and decreases the success of GAs dramatically [19]. Linkage learning capability is therefore a key that makes GAs competent [4].

EDAs, a branch of GAs, do not adopt the crossover operator and the mutation operator. The core idea behind EDAs is to build a probabilistic model that represents the promising solutions found so far. The offspring is then generated based on the model. The way that EDAs model the linkage significantly influences the complexity and the performance.

ECGA [5], one of EDAs, models the linkage as groups of variables and assumes that each group is mutually independent. This model, known as the marginal product model (MPM), consists of two components: (1) a partition that defines mutually independent groups over all variables, and (2) a probabilistic distribution for each group. Assumptions behind ECGA are that a good probability distribution is equivalent to the linkage learning and the 'good' distribution is based on two criteria. The criteria are the compressed representation of the population under the given distribution, and the distribution's representation given the problem's encoding. These two criteria form the combined complexity criteria (CCC), which is expressed as

$$CCC = N \sum_i Entropy(M_i) + \log_2(N+1) \sum_i (2^{S_i} - 1), \quad (1)$$

where N is the population size and the i-th group of an MPM has S_i variables with marginal distribution M_i over this group. $Entropy(M_i)$ is defined as $\sum_k -p_k \log_2(p_k)$, where p_k is the probability of observing the k-th outcome. The entropy of each group means the average number of bits it takes to represent these S_i genes in the population with optimal compression. Therefore, the first term is named the compressed population complexity. The second term represents the memory required to store the MPM structure. It is called the model complexity because, given the MPM structure, one has to record $(2^{S_i} - 1)$ frequency values, and each value requires $\log_2(N+1)$ bits. The sum of two terms can be regarded as the description length of the MPM model. To prevent from over-fitting, the minimum description length principle is taken under the philosophy of Occam's razor. To sum up, the overall objective is to find an MPM model that minimizes CCC to represent the selected population. ECGA uses a greedy algorithm as the search approach to decide the MPM model. The algorithm for model building is listed as Algorithm 1. The complexity of the model-building algorithm is $\Theta(\ell^3)$, where ℓ is the problem size. Using a cache structure that records CCC for each pair of groups, the complexity for model building is reduced to $\Theta(\ell^2 \log \ell)$, as implemented by Lobo *et al.* [9]

Algorithm 1: Model building in ECGA

```
1 begin
2     Each variable is mutually independent. MPM has ℓ
      groups, each group contains one variable, where ℓ is
      the problem size.
3     while Exist a pair that reduces CCC do
4         Greedily find the best pair that reduces CCC
          most
5         Merge the best pair
```

3.2 Previous Work on ECGA with Parallelism

Verma *et al.* [20] divided ECGA into three MapReduces [3]. The first Map phase evaluated the population and finished selection in the Reduce phase. In the model-building step, they partitioned individuals among multiple mappers. Each mapper calculated the marginal probability for every possible pairs of groups and sent the values to the reducers. A single reducer aggregated all the marginal probabilities, greedily found the best pair to merge and updated the model to a file for next model-building MapReduce. After building the model, the third MapReduce was applied to generate offspring. They performed the MapReduce-based ECGA on a cluster of 62 nodes, each with dual Intel Quad cores (8 cores), 16GB RAM and 2TB hard disks. The implementation scaled up to the 1,024-bit trap problem [4] with order 4 under the constraint of memory required to maintain marginal probability. However, the speedup between the MapReduce-based ECGA with cache structure and the sequential version of ECGA with cache structure was not mentioned in the paper.

4. CUDA-BASED ECGA

This section introduces the proposed implementations. A technique that speeds up the process of counting the occurrences of schemata is first described. The way that we utilize the technique to our implementations on GPU and the algorithm of our implementations are then introduced.

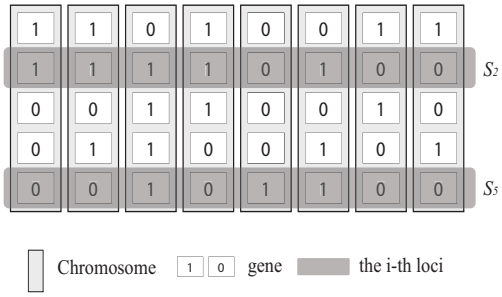


Figure 2: Counting schemata in the second and the fifth loci for the population

4.1 A Table-look-up Method to Speed Up Counting Distribution

A bit-wise table-look-up method is utilized in our implementation to count the occurrences of schemata in the population. For convenience, we call the method FastCount. The core of FastCount is a pre-calculated counting table. The counting table of x-bit binary string records the number of ones in each enumerated value that the x-bit binary string generates. For example, 2-bit binary string generates $\{0, 1, 2, 3\}$, and the corresponding ones are stored to the table as $\{0, 1, 1, 2\}$. For convenience, the table is called unity table afterwards.

The method is applied to EDAs as the following example. Suppose that we want to get the distribution of the second and the fifth loci from a population of size 8 as illustrated in Figure 2. Alleles for the second and the fifth loci among population form two bit stings: $S_2 = 11110100$ and $S_5 = 00101100$. Assume that the values from the i-th position of the two bit-strings form a pair P_i. For example, P_3 is $(1, 1)$. We use n_{xy} to denote the occurrences of the pair (x, y) in the population. Because each bit from $(\neg S_2 \& \neg S_5)$ is 1 only when $(x = 0 \ \& \ y = 0)$, n_{00} can be counted from the resulting bit-string of $(\neg S_2 \& \neg S_5)$. The result of $(\neg S_2 \& \neg S_5)$ is 00000011, or 3 in decimal. By looking up the unity table, we know that the number of ones in the resulting bit-string is 2. Similarly, n_{01} is derived from $(\neg S_2 \& S_5)$, and n_{10} is derived from $(S_2 \& \neg S_5)$. For a larger population, the occurrences can be accumulated from the results of every eight individuals.

We utilize this technique to ECGA when computing compressed population complexity. In the implementation, the selected population is transformed into unsigned int array, with each value representing 32 bits for a certain locus among the 32 individuals. The unity table in CPU is for a 16-bit binary string, which implies that we count the unity from 16 individuals at a time. For memory reason, the unity table in GPU is for a 8-bit binary string.

4.2 gECGA: CUDA-based Implementation

This section introduces our first implementation on GPU, which is called gECGA for convenience. The entire flow chart of gECGA is illustrated in Figure 3, and the overall kernel of the model building is listed in Algorithm 2. The section is followed by the redesign of the cache and the model, the allocation of the memory space, tasks for each thread block and how we update the cache and the model.

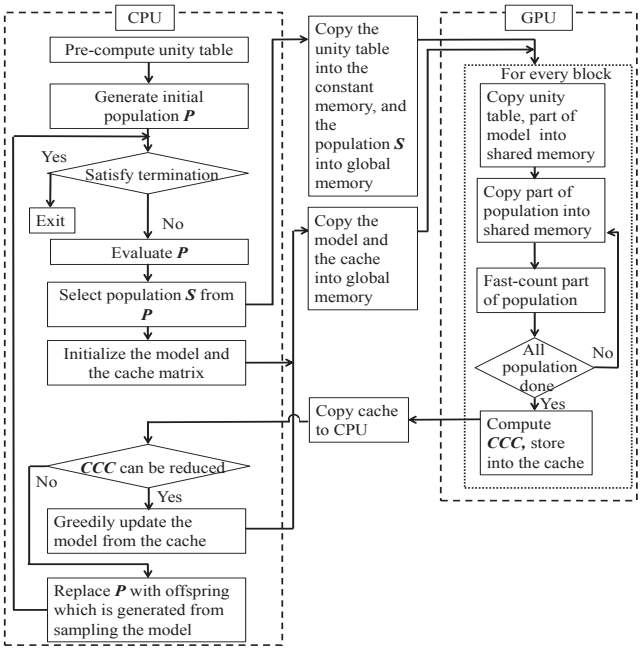


Figure 3: Block diagram of gECGA

Global Memory	Constant Memory	Shared Memory
entire population entire model matrix entire cache matrix status array	unity table	unity table merged group model part of population count array

Table 2: Memory space allocation of gECGA

4.2.1 Cache and model structure

Due to the separated memory spaces of a CPU and a GPU, we need to transfer the data from CPU to GPU. Harik *et al.* mentioned that ECGA could be optimized significantly by caching delta values for all pair combinations at each step [5]. The implementation can be found in [9]. In the implementation, the cache and the MPM model are divided into several C structures. However, passing structure to GPU is troublesome, let alone passing structure inside another structure. As the result, we redesign the cache and the MPM model.

The cache and the MPM model we use are simply two matrices. The cache is an ℓ-by-ℓ matrix, where ℓ is the problem size. Entry (i, j) stores CCC of a group which is merged from $group_i$ and $group_j$. If $i = j$, the entry stores CCC of the $group_i$. The loci in each group are stored in the model matrix. The model is an ℓ-by-L_{row} matrix, where $L_{row} = MAX_L + 1$ and MAX_L is hypothetically the maximum size of a group. L_{row} is equal to $MAX_L + 1$ because the first column stores the size of each group. In all the experiments, MAX_L is set to 10. In addition to the first column, the i-th row of the model matrix stores the loci inside $group_i$. We simultaneously initialize CCC for each group and copy the selected population to GPU.

4.2.2 Memory space allocation

Following the design constraints discussed in Section 2.3,

structure	data type	quantity	cost (bytes)
unity table	CHAR	2^8	256
merged group model	INT	L_{row}	44
count array	INT	$2^{MAX_L} - 1$	4092

Table 3: Memory costs except for the population in the shared memory

the strategy of making the global memory coalesced is to store data that might be accessed by the same warps of threads to contiguous memory. Additionally, the strategy of enhancing CGMA is to store those values that will be frequently used to the shared memory. Consequently, processing the global memory once can be applied to several operations.

The overall allocation is listed in Table 2. In Table 2, status array stores which group just being merged in the previous step, and count array stores the number of occurrences of every schemata. The unity table is invariant and is stored to the constant memory instead of the global memory.

In Table 2.3, the shared memory is 6KB per block under full occupancy on a Tesla C2050 card, which means that GPU can only afford 120 individuals for a 50-bit problem. Even if we transform the population into an UNSIGNED INT array, a block can only process $\frac{6000\times64}{8\times50} = 960$ individuals. It is not enough to solve a GA-hard problem like trap [4]. Therefore, separating the population in the shared memory is needed. In addition to the population, Table 3 lists the memory costs required to store the other structures in the shared memory. The sum is 4,392 bytes. As the result, we still have $(6,000 - 4,392)$ bytes per block, and the amount is equal to 12,864 bits. Given a merged group, we only keep the alleles in this group, and the quantity is not greater than MAX_L. The number of individuals a block can cover at once is therefore derived as $\lfloor\frac{12,864}{MAX_L}\rfloor = 1,286$. When computing CCC, every block sequentially iterates $\lceil\frac{population\ size}{1,286}\rceil$ times to go over the entire population.

4.2.3 Tasks allocation

We partition the model-building process to coarse subproblems that are solved independently in parallel by blocks of threads. We also partition each subproblem to finer pieces that are solved cooperatively in parallel by all threads within the block as the programming guide [16] suggests.

Each iteration of finding the group to merge greedily is called a model-searching step. In every model-searching step, a $|g|$-by-$|g|$ grid which consists of thread blocks is lunched, where $|g|$ is the number of groups in the current MPM model. The (x, y)-th block in the grid calculates CCC of the merging group from $group_i$ and $group_j$. Threads within the block cooperatively finish the following tasks in parallel:

1. Copy the entire unity table from the constant memory to the shared memory.
2. Copy the loci in the merged group from the model matrix in the global memory to the shared memory.
3. Initialize an array to store the distribution.
4. Copy alleles in these loci in the next 1,286 individuals from the global memory to the shared memory.
5. Sequentially count each enumerated value in these individuals by FASTCOUNT.
6. Repeat Step 4 and 5 until the whole population is covered.

When the step of counting the occurrences of schemata is done, one thread in the block calculates the resulting CCC and updates the value to the cache matrix in the global memory. After all the blocks finish updating the cache, the cache matrix is copied back to CPU. The entire algorithm is listed in Algorithm 2.

Algorithm 2: CUDA CCC kernel

```
1  Get the block coordinate (x, y) in the grid of threads
   CUDA launch. The (x, y)-block in the grid is
   responsible to calculate CCC for the merged group
   group_xy from group_x and group_y.
2  tIdx ← the index of the thread in the block.
3  L ← (size of group_x + size of group_y)
4  if BorderCheck(x, y, L) = False then
5      return
6  CopyToShared(x, y, L, tIdx)
7  Synchronize threads in the block.
8  for each partition of population do
9      Parallely copy part of population from the global
       memory to the shared memory.
10     Synchronize threads in the block.
11     Parallely count the occurrences of schemata in the
       partition and store to distribution array.
12     Synchronize threads in the block.
13 if tIdx = 1 then
14     Calculate CCC from the distribution array and
       store to cache.
15 ---------------------------------------------------
16 Function BorderCheck(x, y, L) begin
17     if (x ≥ y) ∨ group_x is deleted ∨ group_y is deleted
       ∨(L > MAX_L) then
18         return False
19     else
20         return True
21 Function CopyToShared(x, y, L, tIdx) begin
22     Copy the tIdx element of the unity table from the
       constant memory to the shared memory
23     if tIdx < L then
24         Copy the tIdx index in the group_xy from the
           global memory to the shared memory
25     i ← tIdx
26     for i < 2^L do
27         i-th element of the distribution array ← zero
           i ← i+total thread number in the block
```

4.2.4 Update cache and model

The updating step is illustrated in Figure 4. Assume that $\{1, 2, 3, 4\}$ is the merged group which minimizes the description length of the model. Numbers in the braces represent the loci in the group. The cache is updated by replacing the merged group with the last group in the current model. Steps (a) and (b) in the figure illustrate this goal. In step (c), we need to update CCC of $\{1, 2, 3, 4\}$ to the $(0, 0)$-entry, which stores CCC of $\{1, 2\}$ originally. When updating the model matrix, the group with the greater index in the merged group appends to the one with the less index, and the last row of the model replaces the row which

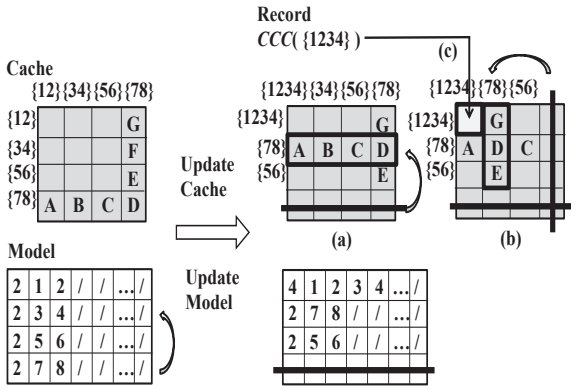


Figure 4: Update cache and model. The model is updated from $\{\{1,2\},\{3,4\},\{5,6\},\{7,8\}\}$ to $\{\{1,2,3,4\},\{5,6\},\{7,8\}\}$.

represents the merged group. The whole updating step is completed in CPU.

4.3 GM Search: The Modified Model-searching Algorithm

In the original ECGA, more than $\frac{\ell(\ell-1)}{2}$ CCC values are calculated in every generation, where ℓ is the problem size. However, for each model-searching step, only one CCC value is utilized from the greedy process. Our method records a group that minimizes the description length for each specific group, instead of finding one among all pairwise candidates greedily. After recording, pairs of groups merge only if they are the best companions to minimize the description length for each other. Therefore, we call this method the greedy mating (GM) search. In each model-searching step, at most half of the groups are merged to the other groups. As the result, GM search needs fewer model-searching steps to find the final model than the original greedy algorithm.

5. EXPERIMENTS

In this section, the hardware specification and general experiment setting are first described. The speedup from FastCount, gECGA and gECGA with GM search are then shown. The scalability of our implementation is also discussed.

5.1 Hardware Specification

We conduct the experiments on a computer with an 8 cores Intel XEON W3530 CPU at 2.8 GHz, and a NVIDIA Tesla C2050 GPU with 3 GB of DDR5 global memory. The GPU card has 448 cores at 1.15GHz. Cores cluster to 14 SMs. The operating system is Gentoo with kernel version 3.4.9. The driver version is NVIDIA-DRIVERS 295.71 with CUDA toolkit 4.0.

5.2 General Experiment Setting

The baseline comparison is the implementation from Lobo *et al.* [9] The test problem is the concatenated traps [4], where the order is 5. The fitness function of each subproblem is defined as

$$f_{trap5}(x) = \begin{cases} 5 & \text{if } x = 5 \\ 4 - x & \text{otherwise.} \end{cases} \quad (2)$$

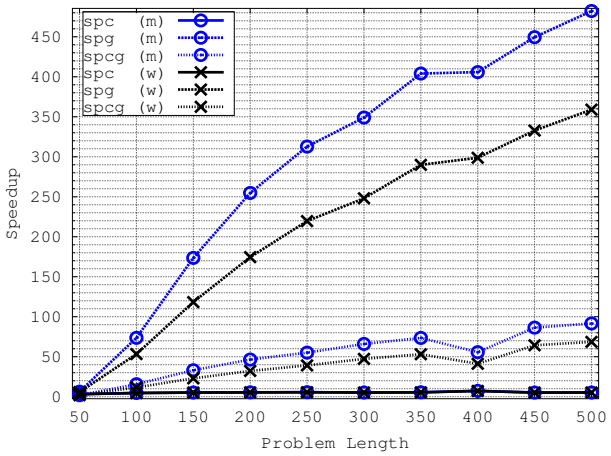


Figure 5: Speedup of FastCount on CPU is denoted as *spc*, speedup of gECGA is denoted as *spg*, and speedup between gECGA and FastCount on CPU is denoted as *spcg*. For the notation after speedup, (m) represents the duration for the entire model-building process, and (w) represents the duration for the whole ECGA process.

The test problem is called (m,k)-trap afterward, where m is the number of subproblems, and k is the order of a subproblem. The minimum population size required to correctly solve at least $m-1$ subproblems is obtained from the average over 30 independent bisection [18] runs, which is listed in Table 5. For problems with ℓ greater than 350, we use the theoretical values instead of running bisection method. Tournament selection is used, and the selection pressure is 8. Previous population is fully replaced by the offspring. The program terminates when all the individuals converge. The unity table on CPU is for 16-bit binary string while the unity table on GPU is for 8-bit binary string due to the memory limit. We assume that the maximum size of a group (MAX_L) is 10, which is also applied to the baseline implementation. Finally, the number of threads per block is set to 256 in the entire experiment.

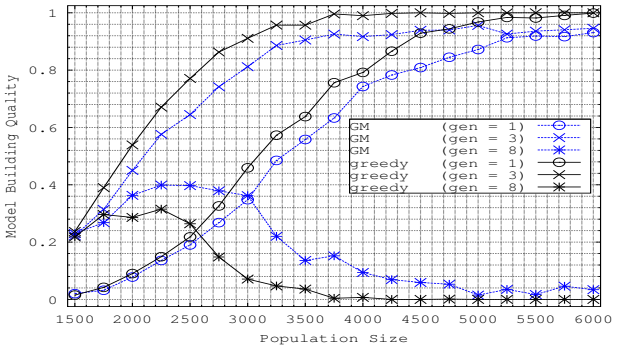
5.3 Speedups

We show the speedups in Figure 5. The speedup between FastCount on CPU and baseline is defined as the ratio of execution time of baseline over the execution time of FastCount method on CPU, which is denoted as *spc*. Similarly, *spg* denotes the speedup between gECGA and baseline, which is defined as the ratio of execution time of baseline over the execution time of gECGA. The speedup between gECGA and FastCount on CPU is denoted as *spcg*. Two different execution periods are shown in Figure 5. The first one is the speedup for model building, and the second one is the speedup for the whole execution time to solve the problem. Each point in the figure is an average over 30 independent runs.

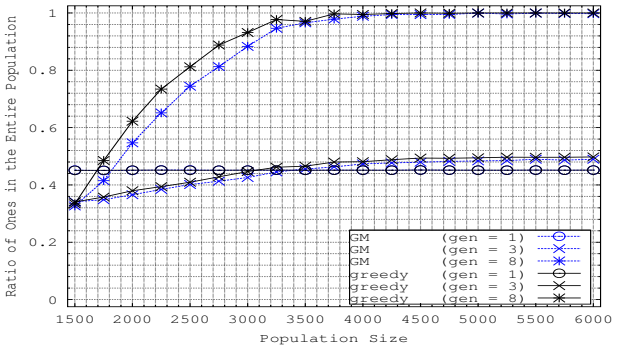
Although FastCount speeds up the counting step, the complexity of model building does not change. We still need to calculate CCC from the resulting distribution after FastCount. Consequently, the overall speedup from FastCount is limited, which is shown as *spc*. The speedups from

structure	data type	cost (byte)
entire population	UNSIGNED INT	$8(\frac{N\ell}{64})$
model matrix	INT	$4\ell L_{row}$
cache matrix	INT	$4\ell^2$
status array	BOOL	ℓ

Table 4: Global memory costs



(a) Model-building quality



(b) Ratio of ones in the population

Figure 6: Model-building quality and ratio of ones in the population for the original ECGA and ECGA with GM search under different population size and different generations (gen).

gECGA can be seen in *spg* and *spcg*. We show the speedup on problems with ℓ up to 500.

Figure 5 shows that gECGA speeds up the model-building process 482.28 times on 500-bit problem, which results to a speedup of 358.97 times faster to solve the problem. The corresponding *spcg* is 68.56, which means that the implementation for FastCount on CUDA is 68.56 times faster than FastCount on CPU using single core.

5.4 Scalability

The only scalability constraint is that the memory which the entire population occupies should be less than the global memory. The shared memory is not a constraint in our implementations. For a Tesla C2050 card, a CUDA program can utilize at most 2.687 GBs of global memory. In Table 2, four structures need to be copied to the global memory. Assume that the population size is N, and the problem size is ℓ. The global memory costs are listed in Table 4. Using the theoretical population size [22], one C2050 GPU card can afford to solve a (2040, 5)-trap problem, where $\ell = 9,800$ and $N = 1,912,315$.

ℓ	50	100	150	200	250	300	350
greedy	2376	6670	11781	17160	21907	29890	36992
GM	2616	6670	12248	19008	24667	30530	39296

Table 5: Population size required to solve $m-1$ subproblems for ECGA with greedy search and ECGA with GM search respectively.

5.5 Experiments of GM Search

The speedup of GM search is not shown in Figure 5 because we suspect that the model-building quality in the new approach is lower than the original method. It is unfair to compare the speedup between the two methods without discussing the model-building quality first. The model-building quality is defined as the number of linkage group that is completely identical with the definition of the subproblem over the number of the subproblems. Figure 6(a) shows the model-building quality under different population sizes and different generations when solving a (10, 5)-trap. The third generation is the generation that achieves the maximum average model quality. The eighth generation is the last generation that program terminates. The figure shows that the overall model-building quality of ECGA with greedy search is greater than with GM search except for the last generation. A possible reason is that the relation between genes is hard to detect when the population is about to converge, and the model-building quality therefore drops in the last generations. Figure 6(b) presents the corresponding ratio of ones in the population after selection. The figure indicates that ECGA with greedy search converges earlier than with GM search, and the model-building quality might therefore drops more quickly.

Due to the loss of model-building quality, the minimum population sizes required to solve the problems are greater, which are listed in Table 5. The speedup from GM search is shown in Figure 7. The population sizes for gECGA with greedy search and gECGA with GM search are different. Although gECGA with GM search needs larger population, the execution time for gECGA with GM search is less than gECGA with the original greedy search.

6. CONCLUSION

In this paper, two implementations were proposed to speed up model building for ECGA. The first implementation was algorithmically identical with ECGA, but solved the problem with a speedup of 358.97 compared with the baseline implementation on a 500-bit trap problem with order 5. The key was to utilize thread blocks in CUDA by separating the model building in ECGA to coarse subproblems that were solved independently in parallel by blocks of threads. Furthermore, each subproblem was separated to finer pieces so that it was solved cooperatively in parallel by all threads within the block. On the basis of the first implementation, our second implementation modified the model building in ECGA. The second implementation solved the same problem with a speedup of 505.6 times faster compared with the baseline. The two implementations scaled up to 9,800-bit trap problem with order 5 on one single Tesla C2050 GPU card.

EDAs have shown the ability to solve many complex problems, and GPUs solve massively parallel computation tasks under relatively lower price than CPUs. To extend the ap-

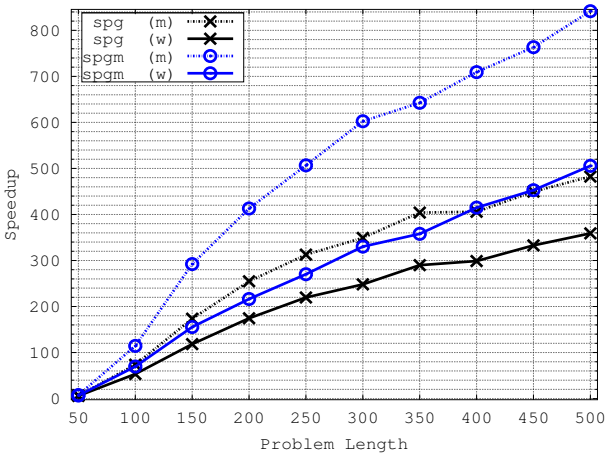


Figure 7: Speedup of gECGA is denoted as *spg*, and speedup of gECGA wit GM search is denoted as *spgm*. For the notation after speedup, (m) represents the duration for the entire model-building process, and (w) represents the duration for the whole ECGA process.

plicability of EDAs to large-scale problems, we believe that EDA researchers need to bear in mind the utilization of GPU features during their designs.

ACKNOWLEDGMENTS

This work was sponsored by the National Science Council of Taiwan under grant NSC-101-2221-E-002-198. The authors would also like to thank Maggie Man-Tin Yung for writing consultation.

7. REFERENCES

[1] S. Baluja. Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning. Technical report, Carnegie Mellon University, Pittsburgh, PA, 1994.

[2] E. Cantú-Paz. *Efficient and accurate parallel genetic algorithms*. Kluwer Academic Publishers, Boston, MA, 2000.

[3] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[4] D. E. Goldberg. *The design of innovation: Lessons from and for competent genetic algorithms*. Kluwer Academic Publishers, Boston, MA, 2002.

[5] G. R. Harik, F. G. Lobo, and K. Sastry. *Linkage Learning via Probabilistic Modeling in the Extended Compact Genetic Algorithm (ECGA)*. 2006.

[6] J. H. Holland. *Adaptation in natural and artificial systems*. Ann Arbor, MI: University of Michigan Press, 1975.

[7] P. Krömer, V. Snásel, J. Platos, and A. Abraham. Many-threaded implementation of differential evolution for the cuda platform. *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1595–1602, 2011.

[8] P. Larrañaga and J. A. Lozano, editors. *Estimation of Distribution Algorithms: A New Tool for Evolutionary Computation*. Kluwer Academic Publishers, Boston, MA, 2002.

[9] F. Lobo, K. Sastry, and G. Harik. Extended compact genetic algorithm in C++: Version 1.1. IlliGAL Report No. 2006012, University of Illinois at Urbana-Champaign, 2006.

[10] H. Mühlenbein and G. Paaß. From recombination of genes to the estimation of distributions I. Binary parameters. In *Parallel Problem Solving from Nature*, pages 178–187. Springer-Verlag, 1996.

[11] A. Munawar, M. Wahib, M. Munetomo, and K. Akama. Hybrid of genetic algorithm and local search to solve max-sat problem using nvidia cuda framework. *Genetic Programming and Evolvable Machines*, 10(4):391–415, 2009.

[12] A. Munawar, M. Wahib, M. Munetomo, and K. Akama. Theoretical and empirical analysis of a gpu based parallel bayesian optimization algorithm. In *Parallel and Distributed Computing, Applications and Technologies, 2009 International Conference on*, pages 457–462. IEEE, 2009.

[13] L. Mussi, F. Daolio, and S. Cagnoni. Evaluation of parallel particle swarm optimization algorithms within the $CUDA^{TM}$ architecture. *Information Sciences*, 181(20):4642–4657, 2011.

[14] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[15] C. Nvidia. C best practices guide. *NVIDIA, Santa Clara, CA*, 2012.

[16] C. Nvidia. CUDA C Programming Guide 4.2. *NVIDIA, Santa Clara, CA*, 2012.

[17] M. Pelikan, D. E. Goldberg, and E. Cantú-Paz. BOA: The Bayesian optimization algorithm. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-1999)*, I:525–532, 1999.

[18] K. Sastry. Evaluation-relaxation schemes for genetic and evolutionary algorithms. Master thesis, University of Illinois at Urbana-Champaign, Urbana, IL, 2002.

[19] D. Thierens. Scalability problems of simple genetic algorithms. *Evolutionary computation*, 7(4):331–352, 1999.

[20] A. Verma, X. Llorà, S. Venkataraman, D. E. Goldberg, and R. H. Campbell. Scaling eCGA model building via data-intensive computing. *Urbana*, 51:61801, 2010.

[21] P. Vidal and E. Alba. Cellular genetic algorithm on graphic processing units. *Nature Inspired Cooperative Strategies for Optimization (NICSO 2010)*, pages 223–232, 2010.

[22] T.-L. Yu, K. Sastry, D. E. Goldberg, and M. Pelikan. Population sizing for entropy-based model building in discrete estimation of distribution algorithms. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2007)*, pages 601–608, 2007.